CHAPTER 13

Statistics

Statistics has long been a field of mathematics that is relevant to practically all applied disciplines of science and engineering, as well as business, medicine, and other fields where data is used for obtaining knowledge and making decisions. With the recent proliferation of data analytics, there has been renewed interest in statistical methods. But computer-aided statistics has a long history, and it is a field traditionally dominated by domain-specific software packages and programming environments, such as the S language and, more recently, its open source counterpart: the R language. The use of Python for statistical analysis has increased rapidly over the last several years, and there is now a mature collection of statistical libraries for Python. With these libraries, Python can match the performance and features of domain-specific languages in many areas of statistics, albeit not all, while also providing the unique advantages of the Python programming language and its environment. The Pandas library discussed in Chapter 12 is an example where traditional statistical software influenced the Python community, especially with the introduction of the data frame data structure to the Python environment. The NumPy and SciPy libraries provide computational tools for many fundamental statistical concepts, and higher-level statistical modeling and machine learning are covered by the `statsmodels` and `scikit-learn` libraries, which we will see more of in the following chapters.

This chapter focuses on fundamental statistical applications using Python, particularly the `stats` module in SciPy. It discusses computing descriptive statistics, random numbers, random variables, distributions, and hypothesis testing. Statistical modeling and machine-learning applications are covered in the upcoming chapters. Some fundamental statistical functions are also available through the NumPy library, such as its functions and methods for computing descriptive statistics and its module for generating random numbers. The SciPy `stats` module builds on top of NumPy and, for example, provides random number generators with more specialized distributions.

Importing Modules

This chapter mainly works with the `stats` module in SciPy. Following the convention to selectively import modules from SciPy, let's assume that this module and the `optimize` module are imported in the following way.

```
In [1]: from scipy import stats
   ...: from scipy import optimize
```

As usual, we also require the NumPy and the Matplotlib libraries.

```
In [2]: import numpy as np
In [3]: import matplotlib.pyplot as plt
```

R. Johansson, *Numerical Python*, https://doi.org/10.1007/979-8-8688-0413-7_13

For statistical graphs and styling, we use the Seaborn library.

```
In [4]: import seaborn as sns
In [5]: sns.set(style="whitegrid")
```

Review of Statistics and Probability

Let's begin with a brief review of statistics, introducing some of the key concepts and the notation we use in this and the following chapters. Statistics involves collecting and analyzing data to gain insights, draw conclusions, and support decision-making. Statistical methods are necessary when we have incomplete information about a phenomenon. Typically, we have incomplete information because we cannot collect data from all members of a *population* or if there is *uncertainty* in our observations (e.g., due to measurement noise). When we cannot survey an entire population, a randomly chosen *sample* can be studied instead. We can use statistical methods and compute descriptive statistics (parameters such as the mean and the variances) to make inferences about the properties of the entire population (also called *sample space*) systematically and with a controlled risk of errors.

Statistical methods are built on the foundation provided by probability theory, with which we can model uncertainty and incomplete information using probabilistic, random variables. For example, with *randomly* selected samples of a population, we can obtain representative samples whose properties can be used to infer properties of the entire population. In probability theory, each possible outcome for an observation is given a probability, and the probability for all possible outcomes constitutes the probability distribution. Given the probability distribution, we can compute the properties of the population, such as its mean and variance. However, for randomly selected samples, we only know the *expected or average* results.

In statistical analysis, it is important to distinguish between population and sample statistics. Here, we denote population parameters with Greek symbols and parameters of a sample with the corresponding population symbol with the added subscript x (or the symbol used to represent the sample). For example, a population's mean and variance are denoted with μ and σ^2, and the mean and the variance of a sample x are denoted as μ_x and σ_x^2. Furthermore, we denote variables representing a population (random variables) with capital letters, for example, X, and a set of sample elements is denoted with a lowercase letter, for example, x. A bar over a symbol denotes the average or mean, $\mu = \bar{X} = \frac{1}{N}\sum_{i=1}^{N} x_i$ and $\mu_x = \bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$, where N is the number of elements in the population X and n is the number of elements in the sample x. The only difference between these two expressions is the number of elements in the sum ($N \geq n$). The situation is slightly more complex for the variance: the population variance is the mean of the squared distance from the mean, $\sigma^2 = \frac{1}{N}\sum_{i=1}^{N}\left(x_i - \mu\right)^2$, and the corresponding sample variance is $\sigma_x^2 = \frac{1}{n-1}\sum_{i=1}^{n}\left(x_i - \mu_x\right)^2$. In the latter expression, we have replaced the population mean μ with the sample mean μ_x and divided the sum with $n - 1$ rather than n. This is because one degree of freedom has been eliminated from the sample set when calculating the mean μ_x, so when computing the sample variance, only $n - 1$ degrees of freedom remain. Consequently, the way to compute the variance for a population and a sample is slightly different. This is reflected in functions we can use to compute these statistics in Python.

Chapter 2 demonstrated that we can compute descriptive statistics for data using NumPy functions or the corresponding `ndarray` methods. For example, to compute the mean and the median of a dataset, we can use the NumPy functions `mean` and `median`.

```
In [6]: x = np.array([3.5, 1.1, 3.2, 2.8, 6.7, 4.4, 0.9, 2.2])
In [7]: np.mean(x)
Out[7]: 3.1
In [8]: np.median(x)
Out[8]: 3.0
```

Similarly, we can use min and max functions or ndarray methods to compute the minimum and maximum values in the array.

```
In [9]: x.min(), x.max()
Out[9]: (0.90, 6.70)
```

To compute the variance and the standard deviation for a dataset, we use the var and std methods. By default, the population variance and standard deviation formulas are used (i.e., it is assumed that the dataset is the entire population).

```
In [10]: x.var()
Out[10]: 3.07
In [11]: x.std()
Out[11]: 1.7521415467935233
```

However, we can use the ddof argument (delta degrees of freedom) to change this behavior. The denominator in the expression for the variance is the number of elements in the array minus ddof, so to calculate the unbiased estimate of the variance and standard deviation from a sample, we need to set ddof=1.

```
In [12]: x.var(ddof=1)
Out[12]: 3.5085714285714293
In [13]: x.std(ddof=1)
Out[13]: 1.8731181032095732
```

The following sections explore how to use NumPy and SciPy's stats module to generate random numbers, represent random variables and distributions, and test hypotheses.

Random Numbers

The Python standard library contains the random module, which provides functions for generating single random numbers with a few elemental distributions. The random module in the NumPy module provides similar functionality but also offers functions that generate NumPy arrays with random numbers, and it has support for a more comprehensive selection of probability distributions. Arrays with random numbers are often practical for computational purposes, so here we focus on the random module in NumPy, and later also the higher-level functions and classes in scipy.stats, which build on top of and extend NumPy.

Earlier in this book, we used np.random.rand, which generates uniformly distributed floating-point numbers in the half-open interval [0, 1) (i.e., 0.0 is a possible outcome, but 1.0 is not). In addition to this function, the np.random module contains an extensive collection of other functions for generating random numbers that cover different intervals, have different distributions, and take values of different types (e.g., floating-point numbers and integers). For example, the randn function produces random numbers that are distributed according to the *standard normal distribution* (the normal distribution with mean 0 and standard deviation 1), and the randint function generates uniformly distributed integers between a given low (inclusive) and high (exclusive) value[1]. When the rand and randn functions are called without arguments, they produce a single random number.

[1] You can use the np.random.seed function to initiate the random number generator in a state that results in reproducible outcomes. Here np.random.seed(123456789) was used.

```
In [14]: np.random.rand()
Out[14]: 0.532833024789759
In [15]: np.random.randn()
Out[15]: 0.8768342101492541
```

However, passing the shape of the array as arguments to these functions produces arrays of random numbers. For example, here we generate a vector of length 5 using `rand` by passing a single argument 5 and a 2 × 4 array using `randn` by passing 2 and 4 as arguments (higher-dimensional arrays are generated by passing the length of each dimension as arguments).

```
In [16]: np.random.rand(5)
Out[16]: array([ 0.71356403,  0.25699895,  0.75269361,  0.88387918,  0.15489908])
In [17]: np.random.randn(2, 4)
Out[17]: array([[ 3.13325952,  1.15727052,  1.37591514,  0.94302846],
               [ 0.8478706 ,  0.52969142, -0.56940469,  0.83180456]])
```

To generate random integers using `randint` (see also `random_integers`), we need to either provide the upper limit for the random numbers (in which case the lower limit is implicitly zero) or provide both the lower and upper limits. The size of the generated array is specified using the `size` keyword arguments, and it can be an integer or a tuple that specifies the shape of a multidimensional array.

```
In [18]: np.random.randint(10, size=10)
Out[18]: array([0, 3, 8, 3, 9, 0, 6, 9, 2, 7])
In [19]: np.random.randint(low=10, high=20, size=(2, 10))
Out[19]: array([[12, 18, 18, 17, 14, 12, 14, 10, 16, 19],
               [15, 13, 15, 18, 11, 17, 17, 10, 13, 17]])
```

The `randint` function generates random integers in the half-open interval [low, high). To demonstrate that the random numbers produced by `rand`, `randn`, and `randint`, are distributed differently, we can plot the histograms of, let's say, 10,000 random numbers produced by each function. The result is shown in Figure 13-1. We note that the distributions for `rand` and `randint` appear uniform but have different ranges and types. In contrast, the distribution of the numbers produced by `randn` resembles a Gaussian curve centered at zero, as expected.

```
In [20]: fig, axes = plt.subplots(1, 3, figsize=(12, 3))
    ...: axes[0].hist(np.random.rand(10000))
    ...: axes[0].set_title("rand")
    ...: axes[1].hist(np.random.randn(10000))
    ...: axes[1].set_title("randn")
    ...: axes[2].hist(np.random.randint(low=1, high=10, size=10000),
    ...:              bins=9, align='left')
    ...: axes[2].set_title("randint(low=1, high=10)")
```

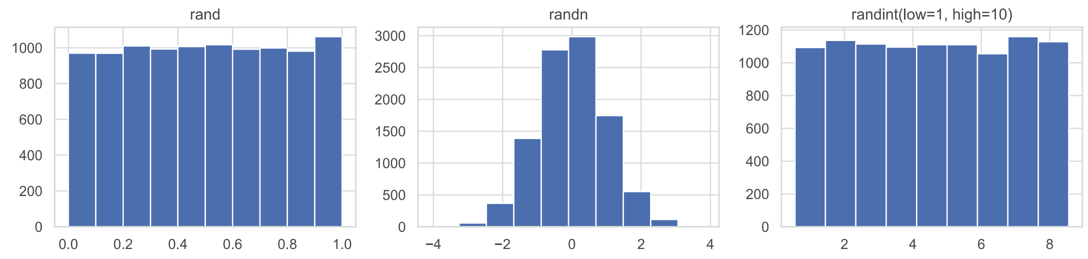

***Figure 13-1.** Distributions for 10,000 random numbers generated by the rand, randn, and randint functions in NumPy's random module*

In statistical analysis, generating a unique list of integers is often necessary. This corresponds to sampling (randomly selecting) items from a set (population) without replacement (so that we do not get the same item twice). We can use the choice function from the NumPy `random` module to generate this type of random number. As the first argument, we can either provide a list (or array) with the population's values or an integer corresponding to the number of elements in the population. As the second argument, we give the number of values to be sampled. The `replace` keyword argument can specify whether the values are sampled with or without replacement, using the Boolean values `True` or `False`. For example, to sample five unique (without replacement) items from the set of integers between 0 (inclusive) and 10 (exclusive), we can use the following.

```
In [21]: np.random.choice(10, 5, replace=False)
Out[21]: array([9, 0, 5, 8, 1])
```

When working with random number generation, it can be useful to *seed* the random number generator. The seed is a number that initializes a random number generator to a specific state. Once seeded with a specific number, it always generates the same sequence of random numbers. This can be useful when testing and reproducing previous results and occasionally in applications requiring reseeding the random number generator (e.g., after having forked a process). To seed the random number generator in NumPy, we can use the `seed` function, which takes an integer as the argument.

```
In [22]: np.random.seed(123456789)
In [23]: np.random.rand()
Out[23]: 0.532833024789759
```

Note that after seeding the random number generator with a specific number, here 123456789, the following calls to the random number generators always produce the same results.

```
In [24]: np.random.seed(123456789); np.random.rand()
Out[24]: 0.532833024789759
```

The seed of the random number generator is a global state of the `np.random` module. A finer level of control of the state of the random number generator can be achieved by using the `RandomState` class, which optionally takes a seed integer as the argument to its initializer. The `RandomState` object keeps track of the state of the random number generator. It allows maintaining several independent random number generators in the same program (which can be useful, e.g., when working with threaded applications). Once a `RandomState` object has been created, we can use the methods of this object to generate random numbers. The `RandomState` class has methods corresponding to the functions available in the `np.random` module. For example, we can use the `randn` method of the `RandomState` class to generate standard normal distributed random numbers.

```
In [25]: prng = np.random.RandomState(123456789)
In [26]: prng.randn(2, 4)
Out[26]: array([[ 2.212902,    2.1283978,   1.8417114,   0.08238248],
                [ 0.85896368, -0.82601643,  1.15727052,  1.37591514]])
```

Similarly, there are methods, `rand`, `randint`, `rand_integers`, and `choice`, which also correspond to the functions in the `np.random` module with the same name. It is considered good programming practice to use a `RandomState` instance rather than directly using the functions in the `np.random` module because it avoids relying on a global state variable and improves the isolation of the code. This is an important consideration when developing library functions that use random numbers but is less critical in smaller applications and calculations.

In addition to the fundamental random number distributions we have looked at so far (discrete and continuous uniform distributions, `randint` and `rand`, and the standard normal distribution, `randn`), there are also functions and `RandomState` methods for many probability distributions that occur in statistics. These include the continuous χ^2 distribution (`chisquare`), the Student's *t* distribution (`standard_t`), and the *F* distribution (`f`).

```
In [27]: prng.chisquare(1, size=(2, 2))
Out[27]: array([[ 0.78631596,  0.19891367],
                [ 0.11741336,  2.8713997 ]])
In [28]: prng.standard_t(1, size=(2, 3))
Out[28]: array([[ 0.39697518, -0.19469463,  1.15544019],
                [-0.65730814, -0.55125015,  0.13578694]])
In [29]: prng.f(5, 2, size=(2, 4))
Out[29]: array([[  0.45471421,  17.64891848,   1.48620557,   2.55433261],
                [  1.21823269,   3.47619315,   0.50835525,   0.70599655]])
```

It also includes the discrete binomial distribution (`binomial`) and the Poisson distribution (`poisson`).

```
In [30]: prng.binomial(10, 0.5, size=10)
Out[30]: array([4, 5, 6, 7, 3, 5, 7, 5, 4, 5])
In [31]: prng.poisson(5, size=10)
Out[31]: array([3, 5, 5, 5, 0, 6, 5, 4, 6, 3])
```

See the docstrings for the `np.random` module, `help(np.random)`, and the `RandomState` class for a complete list of available distribution functions. While it is possible to use the functions in `np.random` and methods in `RandomState` to draw random numbers from many different statistical distribution functions, when working with distributions, there is a higher-level interface in the `scipy.stats` module that combines random number sampling with many other convenient functions for probability distributions. The following section explores this in more detail.

Random Variables and Distributions

In probability theory, the set of possible outcomes of a random process is called the *sample space*. Each element in the sample space (i.e., an outcome of an experiment or an observation) can be assigned a probability, and the probabilities of all possible outcomes define the probability distribution. A *random variable* is a mapping from the sample space to the real numbers or integers. For example, the possible outcomes of a coin toss are head and tail, so the sample space is {head, tail}, and a possible random variable takes the value 0 for head and 1 for tail. There are many ways to define random variables for the possible outcomes of a given random process. Random variables are a problem-independent representation of a random process. It is easier to work with random variables because they are described by numbers instead of outcomes from problem-specific sample spaces. A common step in statistical problem-solving is to map outcomes to numerical values and figure out the probability distribution of those values.

Consequently, a random variable is characterized by its possible values and probability distribution, which assigns a probability for each possible value. Each observation of the random variable results in a random number, and the probability distribution describes the observed values. There are two main types of distributions, discrete and continuous distributions, which are integer-valued and real-valued, respectively. When working with statistics, dealing with random variables is of central importance, and in practice, this often means working with probability distributions. The SciPy `stats` module provides classes for representing random variables with many probability distributions. There are two base classes for discrete and continuous random variables: `rv_discrete` and `rv_continuous`. These classes are not used directly but as base classes for random variables with specific distributions and define a common interface for all random variable classes in SciPy `stats`. A summary of selected methods for discrete and continuous random variables is given in Table 13-1.

Table 13-1. *Selected Methods for Discrete and Continuous Random Variables in the SciPy stats Module*

Methods	**Description**
`pdf`/`pmf`	Probability distribution function (continuous) or probability mass function (discrete)
`cdf`	Cumulative distribution function
`sf`	Survival function (1 – cdf)
`ppf`	Percent-point function (inverse of cdf)
`moment`	Noncentral moments of *n*th order
`stats`	Statistics of the distribution (typically the mean and variance, sometimes additional statistics)
`fit`	Fit distribution to data using a numerical maximum likelihood optimization (for continuous distributions)
`expect`	Expectation value of a function with respect to the distribution
`interval`	The endpoints of the interval that contains a given percentage of the distribution (confidence interval)
`rvs`	Random variable samples Takes as arguments the size of the resulting array of samples
`mean`, `median`, `std`, `var`	Descriptive statistics: mean, median, standard deviation, and the variance of the distribution

There are many classes for the discrete and continuous random variables in the SciPy `stats` module. There are classes for 13 discrete and 98 continuous distributions at the time of writing, including the most encountered distributions (and many less common). For a complete reference, see the docstring for the `stats` module: `help(stats)`. The following explores some of the more common distributions, but the usage of all the other distributions follows the same pattern.

The random variable classes in the SciPy `stats` module have several uses. They are both representations of the distribution, which can be used to compute descriptive statistics and for graphing, and they can be used to generate random numbers following the given distribution using the `rvs` (random variable sample) method. The latter usecase is similar to the `np.random` module was used for earlier in this chapter.

To demonstrate how to use the random variable classes in SciPy `stats`, consider the following example where we create a normal distributed random variable with a mean 1.0 and standard deviation of 0.5.

```
In [32]: X = stats.norm(1, 0.5)
```

Now `X` is an object that represents a random variable, and we can compute descriptive statistics of this random variable using, for example, the `mean`, `median`, `std`, and `var` methods.

```
In [33]: X.mean()
Out[33]: 1.0
In [34]: X.median()
Out[34]: 1.0
In [35]: X.std()
Out[35]: 0.5
In [36]: X.var()
Out[36]: 0.25
```

Noncentral moments of arbitrary order can be computed with the `moment` method.

```
In [37]: [X.moment(n) for n in range(5)]
Out[37]: [1.0, 1.0, 1.25, 1.75, 2.6875]
```

We can obtain a distribution-dependent list of statistics using the `stats` method (here, for a normal distributed random variable, we get the mean and the variance).

```
In [38]: X.stats()
Out[38]: (1.0, 0.25)
```

We can evaluate the probability distribution function, the cumulative distribution function, the survival function, using methods like `pdf`, `cdf`, and `sf`. These all take a value, or an array of values, to evaluate the function.

```
In [39]: X.pdf([0, 1, 2])
Out[39]: array([ 0.10798193,  0.79788456,  0.10798193])
In [40]: X.cdf([0, 1, 2])
Out[40]: array([ 0.02275013,  0.5,        0.97724987])
```

The `interval` method can compute the lower and upper values of *x* such that a given percentage of the probability distribution falls within the interval (lower, upper). This method is helpful in computing confidence intervals and for selecting a range of *x* values for plotting.

```
In [41]: X.interval(0.95)
Out[41]: (0.020018007729972975, 1.979981992270027)
In [42]: X.interval(0.99)
Out[42]: (-0.28791465177445019, 2.2879146517744502)
```

To build intuition for the properties of a probability distribution, it is useful to graph it together with the corresponding cumulative probability function and the percent-point function. To make it easier to repeat this for several distributions, we first create a `plot_rv_distribution` function that plots the result of `pdf` or `pmf`, the `cdf` and `sf`, and `ppf` methods of the SciPy `stats` random variable objects over an interval that contains 99.9% of the probability distribution function. We also highlight the area that contains 95% of the probability distribution using the `fill_between` drawing method.

```
In [43]: def plot_rv_distribution(X, axes=None):
    ...:     """Plot the PDF or PMF, CDF, SF and PPF of a given random variable"""
    ...:     if axes is None:
    ...:         fig, axes = plt.subplots(1, 3, figsize=(12, 3))
    ...:
    ...:     x_min_999, x_max_999 = X.interval(0.999)
    ...:     x999 = np.linspace(x_min_999, x_max_999, 1000)
    ...:     x_min_95, x_max_95 = X.interval(0.95)
    ...:     x95 = np.linspace(x_min_95, x_max_95, 1000)
    ...:
    ...:     if hasattr(X.dist, "pdf"):
    ...:         axes[0].plot(x999, X.pdf(x999), label="PDF")
    ...:         axes[0].fill_between(x95, X.pdf(x95), alpha=0.25)
    ...:     else:
    ...:         # discrete random variables do not have a pdf method,
    ...:         # instead  use pmf:
    ...:         x999_int = np.unique(x999.astype(int))
    ...:         axes[0].bar(x999_int, X.pmf(x999_int), label="PMF")
    ...:     axes[1].plot(x999, X.cdf(x999), label="CDF")
    ...:     axes[1].plot(x999, X.sf(x999), label="SF")
    ...:     axes[2].plot(x999, X.ppf(x999), label="PPF")
    ...:
    ...:     for ax in axes:
    ...:         ax.legend()
```

Next, we use this function to graph a few examples of distributions: the normal distribution, the *F* distribution, and the discrete Poisson distribution. The result is shown in Figure 13-2.

```
In [44]: fig, axes = plt.subplots(3, 3, figsize=(12, 9))
    ...: X = stats.norm()
    ...: plot_rv_distribution(X, axes=axes[0, :])
    ...: axes[0, 0].set_ylabel("Normal dist.")
    ...: X = stats.f(2, 50)
    ...: plot_rv_distribution(X, axes=axes[1, :])
    ...: axes[1, 0].set_ylabel("F dist.")
    ...: X = stats.poisson(5)
    ...: plot_rv_distribution(X, axes=axes[2, :])
    ...: axes[2, 0].set_ylabel("Poisson dist.")
```

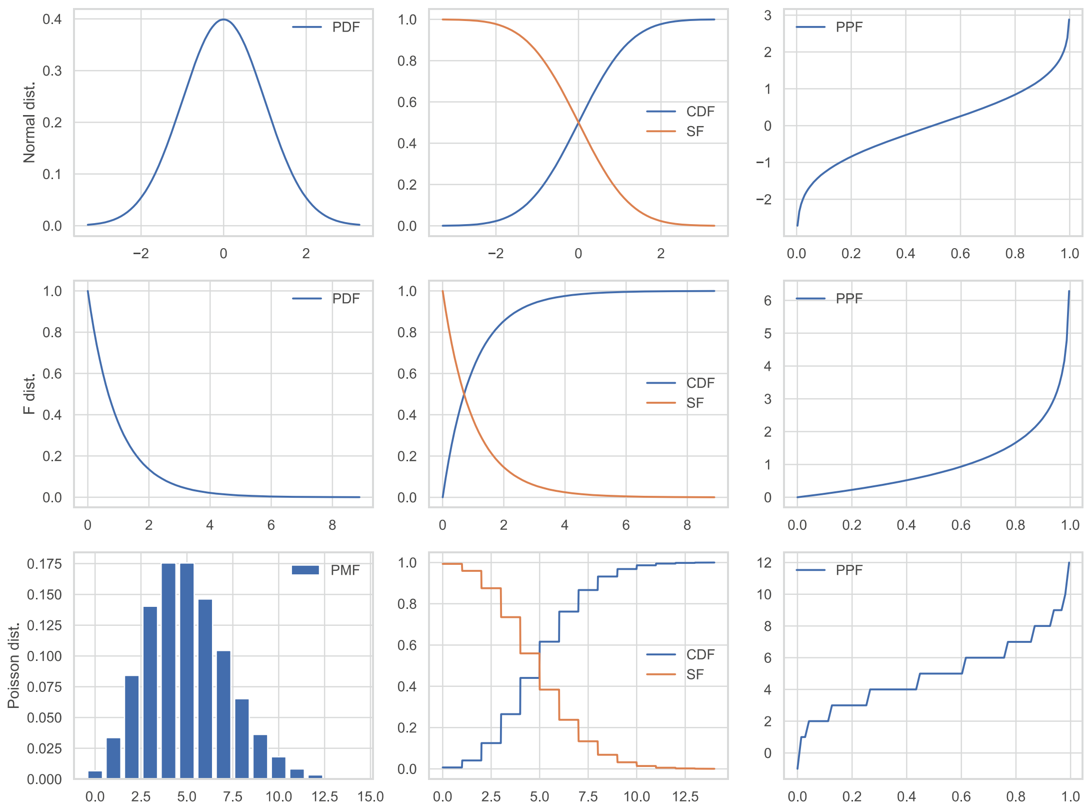

***Figure 13-2.** Examples of probability distribution functions (PDF) or probability mass functions (PMFs), cumulative distribution functions (CDF), survival functions (SF), and percent-point functions (PPF) for a normal distribution (top), an F distribution (middle), and a Poisson distribution (bottom)*

The examples so far have initiated an instance of a random variable class and computed statistics and other properties using method calls. An alternative way to use the random variable classes in SciPy's `stats` module is to use class methods, for example, `stats.norm.mean`, and pass the distribution parameters as arguments (often `loc` and `scale`, as in this example for normally distributed values).

```
In [45]: stats.norm.stats(loc=2, scale=0.5)
Out[45]: (2.0, 0.25)
```

This gives the same result as creating an instance and then calling the corresponding method.

```
In [46]: stats.norm(loc=1, scale=0.5).stats()
Out[46]: (1.0, 0.25)
```

Most methods in the `rv_discrete` and `rv_continuous` classes can be used as class methods in this way.

So far, we have only looked at properties of the distribution function of random variables. Note that although a distribution function describes a random variable, the distribution itself is entirely deterministic. To draw random numbers that are distributed according to the given probability distribution, we can use the rvs (random variable sample) method. It takes as the argument the shape of the required array (can be an integer for a vector or a tuple of dimension lengths for a higher-dimensional array). Here, we use rvs(10) to generate a one-dimensional array with ten values.

```
In [47]: X = stats.norm(1, 0.5)
In [48]: X.rvs(10)
Out[48]: array([2.106451,    2.0641989,   1.9208557,  1.04119124,  1.42948184,
                0.58699179,  1.57863526,  1.68795757, 1.47151423,  1.4239353 ])
```

To see that the resulting random numbers are indeed distributed according to the corresponding probability distribution function, we can graph a histogram of many samples of a random variable and compare it to the probability distribution function. Again, to do this easily for samples of several random variables, we create a plot_dist_samples function. This function uses the interval method to obtain a suitable plot range for a random variable object.

```
In [49]: def plot_dist_samples(X, X_samples, title=None, ax=None):
    ...:     """Plot the PDF and histogram of samples of a continuous
    ...:     random variable"""
    ...:     if ax is None:
    ...:          fig, ax = plt.subplots(1, 1, figsize=(8, 4))
    ...:
    ...:     x_lim = X.interval(.99)
    ...:     x = np.linspace(*x_lim, num=100)
    ...:
    ...:     ax.plot(x, X.pdf(x), label="PDF", lw=3)
    ...:     ax.hist(X_samples, label="samples", density=1, bins=75)
    ...:     ax.set_xlim(*x_lim)
    ...:     ax.legend()
    ...:
    ...:     if title:
    ...:         ax.set_title(title)
    ...:     return ax
```

Note that in this function, we have used the tuple unpacking syntax *x_lim, which distributes the elements in the tuple x_lim to different arguments for the function. In this case, it is equivalent to np.linspace(x_lim[0], x_lim[1], num=100).

Next, we use this function to visualize 2000 samples of three random variables with different distributions: here, we use the Student's *t* distribution, the χ^2 distribution, and the exponential distribution, and the results are shown in Figure 13-3. Since 2000 is a reasonably large sample, the histogram graphs of the samples coincide well with the probability distribution function. The agreement can be expected to be even better with an even larger number of samples.

```
In [50]: fig, axes = plt.subplots(1, 3, figsize=(12, 3))
    ...: N = 2000
    ...: # Student's t distribution
    ...: X = stats.t(7.0)
    ...: plot_dist_samples(X, X.rvs(N), "Student's t dist.", ax=axes[0])
    ...: # The chisquared distribution
    ...: X = stats.chi2(5.0)
    ...: plot_dist_samples(X, X.rvs(N), r"$\chi^2$ dist.", ax=axes[1])
    ...: # The exponential distribution
    ...: X = stats.expon(0.5)
    ...: plot_dist_samples(X, X.rvs(N), "exponential dist.", ax=axes[2])
```

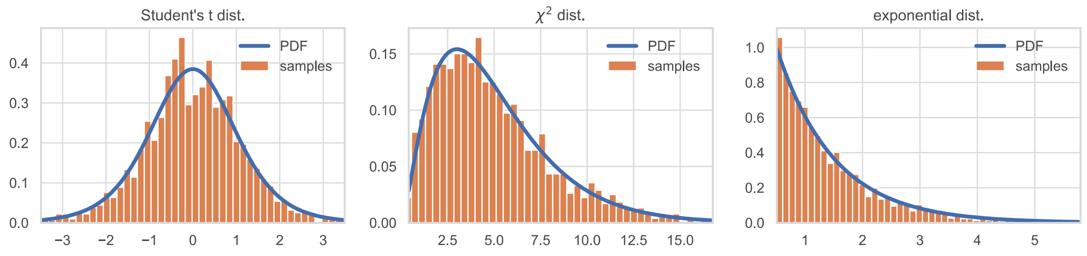

Figure 13-3. *Probability distribution function (PDF) together with histograms of 2000 random samples from the Student's t distribution (left), the χ^2 distribution (middle), and the exponential distribution (right)*

The opposite of drawing random samples from a known distribution function is to fit a given probability distribution with unknown parameters to a set of data points. In such a fit, we typically wish to optimize the unknown parameters to maximize the likelihood of observing the given data. This is called a maximum likelihood fit. Many of the random variable classes in the SciPy `stats` module implement the method `fit` that performs such a fitting to given data. As a first example, consider drawing 500 random samples from the χ^2 distribution with five degrees of freedom (`df=5`) and then refitting the random variables to the χ^2 distribution using the `fit` method.

```
In [51]: X = stats.chi2(df=5)
In [52]: X_samples = X.rvs(500)
In [53]: df, loc, scale = stats.chi2.fit(X_samples)
In [54]: df, loc, scale
Out[54]: (5.2886783664198465, 0.0077028130326141243, 0.93310362175739658)
In [55]: Y = stats.chi2(df=df, loc=loc, scale=scale)
```

The `fit` method returns the maximum likelihood parameters of the distribution for the given data. We can pass those parameters to the initializer of the `stats.chi2` to create a new random variable instance `Y`. The probability distribution of `Y` should resemble the probability distribution of the original random variable `X`. To verify this, we can plot the probability distribution functions for both random variables. The resulting graph is shown in Figure 13-4.

```
In [56]: fig, axes = plt.subplots(1, 2, figsize=(12, 4))
    ...: x_lim = X.interval(.99)
    ...: x = np.linspace(*x_lim, num=100)
    ...:
    ...: axes[0].plot(x, X.pdf(x), label="original")
    ...: axes[0].plot(x, Y.pdf(x), label="recreated")
    ...: axes[0].legend()
    ...:
    ...: axes[1].plot(x, X.pdf(x) - Y.pdf(x), label="error")
    ...: axes[1].legend()
```

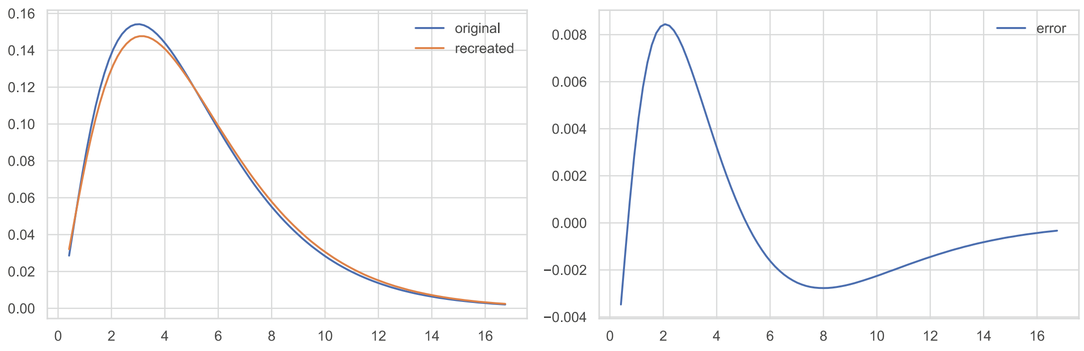

Figure 13-4. *Original and re-created probability distribution function (left) and the error (right) from a maximum likelihood fit of 500 random samples of the original distribution*

This section explored how to use random variable objects from the SciPy stats model to describe random variables with various distributions and how they can be used to compute the given distributions' properties, generate random variable samples, and perform maximum likelihood fitting. The following section explains how to use these random variable objects for hypothesis testing.

Hypothesis Testing

Hypothesis testing is a cornerstone of the scientific method, which requires that claims are investigated objectively and that a claim is rejected or accepted based on factual observations. Statistical hypothesis testing has a more specific meaning. It is a systematic methodology for evaluating whether a claim or a hypothesis is reasonable based on data. As such, it is an important application of statistics. In this methodology, we formulate the hypothesis using a null hypothesis, H_0, which represents the currently accepted state of knowledge, and an alternative hypothesis, H_A, which represents a new claim that challenges the current state of knowledge. The null hypothesis and the alternative hypothesis must be mutually exclusive and complementary so that one and only one of the hypotheses is true.

Once H_0 and H_A are defined, the data that support the test must be collected, for example, through measurements, observations, or a survey. The next step is to find a test statistic that can be computed from the data and whose probability distribution function can be found under the null hypothesis. We can evaluate the data by computing the probability (the *p-value*) of obtaining the observed value of the test statistic (or a more extreme one) using the distribution function implied by the null hypothesis. If the p-value is smaller than a predetermined threshold, known as the significance level, and denoted by α (typically 5% or 1%), we can conclude that the observed data is unlikely to have been described by the distribution corresponding to the null hypothesis. In that case, we can, therefore, reject the null hypothesis in favor of the alternative hypothesis. The steps for carrying out a hypothesis test are summarized in the following list.

1. Formulate the null hypothesis and the alternative hypothesis.
2. Select a test statistic such that its sampling distribution under the null hypothesis is known (exactly or approximately).
3. Collect data.
4. Compute the test statistics from the data and calculate its p-value under the null hypothesis.
5. If the p-value is smaller than the predetermined significance level α, we reject the null hypothesis. If the p-value is larger, we fail to reject the null hypothesis.

Statistical hypothesis testing is a probabilistic method, which means we cannot be certain whether to reject or not reject the null hypothesis. There can be two types of error: we can mistakenly reject the null hypothesis when it should not be rejected, and we can fail to reject the null hypothesis when it should be rejected. These are called type I and type II errors, respectively. By choosing the required significance level, we can balance the trade-off between these two types of error.

The most challenging step in the method outlined in the preceding section is knowing the sampling distribution of the test statistics. Fortunately, many hypothesis tests fall into a few standard categories for which the probability distributions are known. A summary and overview of common hypothesis test cases and the corresponding distribution of their test statistics are given in Table 13-2. For motivations for why each of these tests is suitable for stated situations and the complete set of conditions for the validity of the tests, see statistics textbooks such as Wasserman (2004) or Rice (1995). The docstring for each listed function in the SciPy `stats` module also contains further information about each test.

***Table 13-2.** Summary of Common Hypothesis Test Cases with the Corresponding Distributions and SciPy Functions*

Null Hypothesis	Distributions	SciPy Functions for Test
Test if the mean of a population is a given value.	Normal distribution (`stats.norm`), or Student's *t* distribution (`stats.t`)	`stats.ttest_1samp`
Test if the means of two random variables are equal (independent or paired samples).	Student's *t* distribution (`stats.t`)	`stats.ttest_ind`, `stats.ttest_rel`
Test goodness of fit of a continuous distribution to data.	Kolmogorov-Smirnov distribution	`stats.kstest`
Test if categorical data occur with given frequency (sum of squared normally distributed variables).	χ^2 distribution (`stats.chi2`)	`stats.chisquare`
Test for the independence of categorical variables in a contingency table.	χ^2 distribution (`stats.chi2`)	`stats.chi2_contingency`
Test for equal variance in samples of two or more variables.	*F* distribution (`stats.f`)	`stats.barlett`, `stats.levene`
Test for noncorrelation between two variables.	Beta distribution (`stats.beta`, `stasts.mstats.betai`)	`stats.pearsonr`, `stats.spearmanr`
Test if two or more variables have the same population mean (ANOVA—analysis of variance).	*F* distribution	`stats.f_oneway`, `stats.kruskal`

The following also looks at examples of how the corresponding functions in the SciPy `stats` module can be used to carry out steps 4 and 5 in the preceding procedure: computing a test statistic and the corresponding *p*-value.

For example, a common null hypothesis is a claim that the mean μ of a population is a certain value μ_0. We can then sample the population and use the sample mean $\bar{x}$ to form a test statistic $z = \frac{\bar{x} - \mu_0}{\sigma / \sqrt{n}}$, where n is the sample size. If the population is large and the variance σ is known, then it is reasonable to assume that the test statistic is normally distributed. If the variance is unknown, we can substitute σ^2 with the sample variance σ_x^2. The test statistic then follows the Student's *t* distribution, which approaches the normal distribution in the limit of a large number of samples. Regardless of which distribution is used, we can compute a *p*-value for the test statistics using the given distribution.

As an example of how this type of hypothesis test can be carried out using the functions provided by the SciPy `stats` module, consider a null hypothesis that claims that a random variable *X* has mean $\mu_0 = 1$. Given samples of *X*, we then wish to test if the sampled data is compatible with the null hypothesis. Here, we simulate the samples by drawing 100 random samples from a distribution slightly different than that claimed by the null hypothesis (using $\mu = 0.8$).

```
In [57]: mu0, mu, sigma = 1.0, 0.8, 0.5
In [58]: X = stats.norm(mu, sigma)
In [59]: n = 100
In [60]: X_samples = X.rvs(n)
```

Given the sample data, `X_samples`, next, we need to compute a test statistic. If the population standard deviation σ is known, as in this example, we can use $z = \frac{\bar{x} - \mu_0}{\sigma / \sqrt{n}}$, which is normally distributed.

```
In [61]: z = (X_samples.mean() - mu0)/(sigma/np.sqrt(n))
In [62]: z
Out[62]: -2.8338979550098298
```

If the population variance is unknown, we can use the sample standard deviation instead: $t = \frac{\bar{x} - \mu}{\sigma_x / \sqrt{n}}$. However, in this case, the test statistics *t* follows the Student's *t* distribution instead of the normal one. To compute *t* in this case, we can use the NumPy method `std` with the `ddof=1` argument to compute the sample standard deviation.

```
In [63]: t = (X_samples.mean() - mu0)/(X_samples.std(ddof=1)/np.sqrt(n))
In [64]: t
Out[64]: -2.9680338545657845
```

In either case, we get a test statistic to compare with the corresponding distribution to obtain a *p*-value. For example, for a normal distribution, we can use a `stats.norm` instance to represent a normal distributed random variable, and with its `ppf` method, we can look up the statistics value corresponding to a certain significance level. For a two-sided hypothesis test of significance level 5% (2.5% on each side), the statistics threshold is as follows.

```
In [65]: stats.norm().ppf(0.025)
Out[65]: -1.9599639845400545
```

Since the observed statistic is about –2.83, smaller than the threshold value of –1.96 for a two-sided test with a significance level of 5%, we have sufficient grounds to reject the null hypothesis in this case. We can explicitly compute the *p*-value for the observed test statistics using the `cdf` method (multiplied by two for a two-sided test). The resulting *p*-value is relatively small, which supports rejecting the null hypothesis.

```
In [66]: 2 * stats.norm().cdf(-abs(z))
Out[66]: 0.0045984013290753566
```

If we want to use the *t* distribution, we can use the `stats.t` class instead of the `stats.norm`. After computing the sample mean, $\bar{x}$, only $n - 1$ degrees of freedom (`df`) remain in the sample data. The number of degrees of freedom is an important parameter for the *t* distribution, which we need to specify when we create the random variable instance.

```
In [67]: 2 * stats.t(df=(n-1)).cdf(-abs(t))
Out[67]: 0.0037586479674227209
```

Again, the p-value is very small, suggesting we should reject the null hypothesis. Instead of explicitly carrying out these steps (computing the test statistics, then computing the *p*-value), there are built-in functions in SciPy's `stats` module for carrying out many common tests, as summarized in Table 13-2. For the test used here, we can directly compute the test statistics and the *p*-value using the `stats.ttest_1samp` function.

```
In [68]: t, p = stats.ttest_1samp(X_samples, mu)
In [69]: t
Out[69]: -2.9680338545657841
In [70]: p
Out[70]: 0.0037586479674227209
```

Again, we see that the *p*-value is very small (the same value as in the preceding text) and that we should reject the null hypothesis. Plotting the distribution corresponding to the null hypothesis and the sampled data is also illustrative (see Figure 13-5).

```
In [71]: fig, ax = plt.subplots(figsize=(8, 3))
    ...: sns.histplot(X_samples, kde=True, stat='density', ax=ax)
    ...: x = np.linspace(*X.interval(0.999), num=100)
    ...: ax.plot(x, stats.norm(loc=mu, scale=sigma).pdf(x))
```

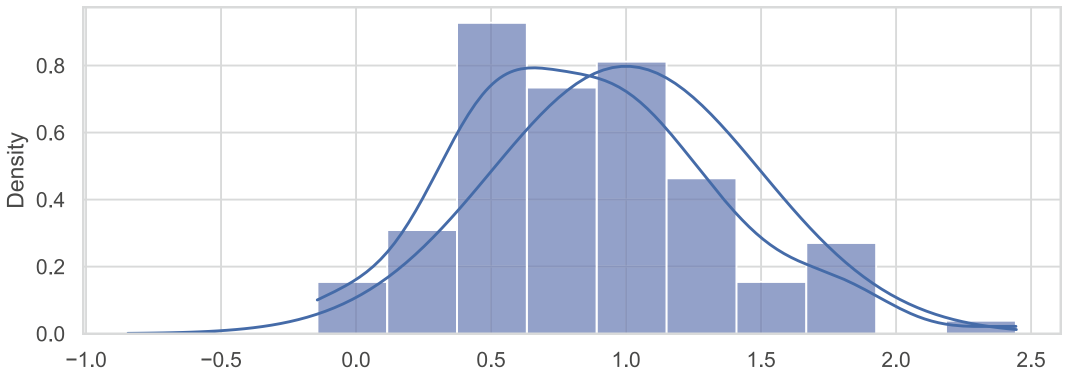

Figure 13-5. *Distribution function according to the null hypothesis (light green) and the sample estimated distribution function (dark blue)*

For another example, consider a two-variable problem where the null hypothesis states that the population means of two random variables are equal (e.g., corresponding to independent subjects with and without treatment). We can simulate this type of test by creating two random variables with normal distribution, with a randomly chosen population means. Here, we select 50 samples for each random variable.

```
In [72]: n, sigma = 50, 1.0
In [73]: mu1, mu2 = np.random.rand(2)
In [74]: X1 = stats.norm(mu1, sigma)
In [75]: X1_sample = X1.rvs(n)
In [76]: X2 = stats.norm(mu2, sigma)
In [77]: X2_sample = X2.rvs(n)
```

We want to evaluate whether the observed samples provide sufficient evidence that the two population means are unequal (rejecting the null hypothesis). For this situation, we can use the *t*-test for two independent samples, which is available in SciPy's `stats.ttext_ind`, which returns the test statistics and the corresponding *p*-value.

```
In [78]: t, p = stats.ttest_ind(X1_sample, X2_sample)
In [79]: t
Out[79]: -1.4283175246005888
In [80]: p
Out[80]: 0.15637981059673237
```

Here, the *p*-value is about 0.156, which is not small enough to support rejecting the null hypothesis that the two means are different. In this example, the two population means are different.

```
In [81]: mu1, mu2
Out[81]: (0.24764580637159606, 0.42145435527527897)
```

However, the samples drawn from these distributions did not statistically prove that these means are different (an error of type II). To increase the power of the statistical test, we would need to increase the number of samples from each random variable.

The SciPy `stats` module contains functions for common types of hypothesis testing (see the summary in Table 13-2), and their use closely follows what was shown in the examples in this section. However, some tests require additional arguments for distribution parameters. See the docstrings for each test function for details.

Nonparametric Methods

So far, we have described random variables with distributions completely determined by a few parameters, such as the mean and the variance for the normal distributions. Given the sampled data, we can fit a distribution function using maximum likelihood optimization with respect to the distribution parameters. Such distribution functions are called *parametric*, and statistical methods based on such distribution functions (e.g., a hypothesis test) are called *parametric methods*. When using those methods, we strongly assume that the given distribution describes the sampled data. An alternative approach to constructing a representation of an unknown distribution function is *kernel-density estimation* (KDE), which can be viewed as a smoothened version of the histogram of the sampled data (see, e.g., Figure 13-6). In this method, the probability distribution is estimated by a sum of the kernel function centered at each data point $\hat{f}(x)=\frac{1}{n.\mathrm{bw}}\sum_{i=0}^{n}K\left(\frac{x-x_i}{\mathrm{bw}}\right)$, where bw is a free parameter known as the bandwidth, and K is the kernel function (normalized to integrate to unity). The bandwidth is an important parameter that defines a scale for the influence of each term in the sum. A too-broad bandwidth gives a featureless estimate of the probability distribution, and a too-small bandwidth gives a noisy, overly structured estimate (see the middle panel in Figure 13-6). Different choices of kernel functions are also possible. A Gaussian kernel is a popular choice because of its smooth shape with local support, and it is relatively easy to perform computations.

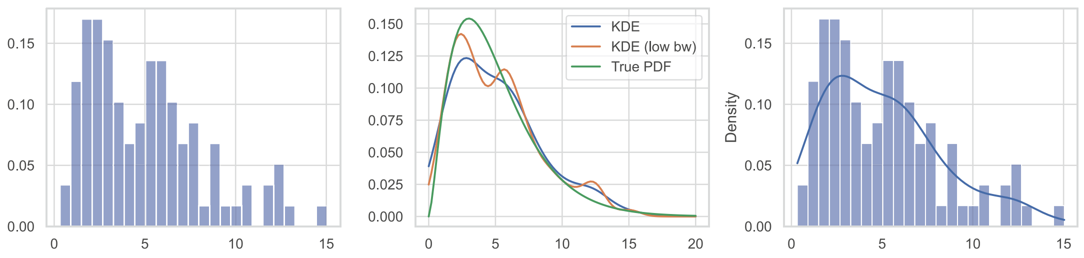

Figure 13-6. *Histogram (left), kernel-density estimation of the distribution function (middle), and both a histogram and the kernel-density estimate in the same graph (right)*

In SciPy, the KDE method using a Gaussian kernel is implemented in the `stats.gaussian_kde` function, which returns a callable object that behaves as and can be used as a probability distribution function. For example, consider a set of samples, `X_samples`, drawn from a random variable *X* with an unknown distribution (here simulated using the χ^2 distribution with five degrees of freedom).

```
In [82]: X = stats.chi2(df=5)
In [83]: X_samples = X.rvs(100)
```

To compute the kernel-density estimate for the given data, call the `stats.guassian_kde` function with the array of sample points as the argument.

```
In [84]: kde = stats.gaussian_kde(X_samples)
```

A standard method for computing a suitable bandwidth is used by default, often giving acceptable results. However, we could also specify a function for computing the bandwidth or directly setting the bandwidth using the `bw_method` argument. To select a smaller bandwidth, we can, for example, use the following.

```
In [85]: kde_low_bw = stats.gaussian_kde(X_samples, bw_method=0.25)
```

The `gaussian_kde` function returns an estimate of the distribution function, which we, for example, can graph or use for other applications. Here, we plot a histogram of the data and the two kernel-density estimates (with default and explicitly set bandwidth). For reference, let's also plot the true probability distribution function for the samples. The result is shown in Figure 13-6.

```
In [86]: x = np.linspace(0, 20, 100)
In [87]: fig, axes = plt.subplots(1, 3, figsize=(12, 3))
    ...: axes[0].hist(X_samples, density=True, alpha=0.5, bins=25)
    ...: axes[1].plot(x, kde(x), label="KDE")
    ...: axes[1].plot(x, kde_low_bw(x), label="KDE (low bw)")
    ...: axes[1].plot(x, X.pdf(x), label="True PDF")
    ...: axes[1].legend()
    ...: sns.distplot(X_samples, bins=25, ax=axes[2])
```

The `seaborn` statistical graphics library provides a convenient function for plotting a histogram and the kernel-density estimation for a set of data: `histplot`. A graph produced by this function is shown in the right panel of Figure 13-6.

Given the kernel-density estimate, we can also use it to generate new random numbers using the `resample` method, which takes the number of data points as arguments.

```
In [88]: kde.resample(10)
Out[88]: array([[1.75376869,  0.5812183,  8.19080268,  1.38539326,  7.56980335,
                1.16144715,  3.07747215,  5.69498716,  1.25685068,  9.55169736]])
```

The kernel-density estimate object does not directly contain methods for computing the cumulative distribution functions (CDF) and its inverse, the percent-point function (PPF). However, several methods exist for integrating the kernel-density estimate of the probability distribution function. For example, for a one-dimensional KDE, we can use the `integrate_box_1d` to obtain the corresponding CDF.

```
In [89]: def _kde_cdf(x):
    ...:     return kde.integrate_box_1d(-np.inf, x)
In [90]: kde_cdf = np.vectorize(_kde_cdf)
```

We can use the SciPy `optimize.fsolve` function to find the inverse (the PPF).

```
In [91]: def _kde_ppf(q):
    ...:     return optimize.fsolve(
    ...:         lambda x, q: kde_cdf(x) - q, kde.dataset.mean(), args=(q,))[0]
    ...:
In [92]: kde_ppf = np.vectorize(_kde_ppf)
```

With the CDF and PPF for the kernel-density estimate, we can, for example, perform statistical hypothesis testing and compute confidence intervals. For example, using the `kde_ppf` function defined in the preceding section, we can compute an approximate 90% confidence interval for the mean of the population from which the sample was collected.

```
In [93]: kde_ppf([0.05, 0.95])
Out[93]: array([  0.39074674,  11.94993578])
```

As illustrated with this example, once we have a KDE that represents the probability distribution for a statistical problem, we can proceed with many of the same methods as we use in parametric statistics. The advantage of nonparametric methods is that we do not necessarily need to make assumptions about the shape of the distribution function. However, their statistical power is lower because nonparametric methods use less information (weaker assumptions) than parametric methods. Therefore, if we can justify using a parametric method, then using it is the best approach. Nonparametric methods offer a versatile generic approach that we can fall back on when parametric methods are not feasible.

Summary

This chapter explored how NumPy and the SciPy `stats` module can be used in basic statistical applications, including random number generation, for representing random variables and probability distribution functions, maximum likelihood fitting distributions to data, and using probability distributions and test statistics for hypothesis testing. We also briefly looked at kernel-density estimation of an unknown probability distribution as an example of a nonparametric method. The concepts and methods discussed in this chapter are fundamental building blocks for working with statistics, and the computational tools introduced here also provide a foundation for many statistical applications. The upcoming chapters build on what has been discussed here and explore statistical modeling and machine learning in more depth.

Further Reading

Good introductions to the fundamentals of statistics and data analysis are given in *Mathematical Statistics and Data Analysis* by J. A. Rice (Duxbury Press, 1995) and *All of Statistics* Wasserman (Springer, 2004). A computationally oriented introduction to statistics is given in *Mathematical Statistics and Data Analysis* by P. Dalgaard (Springer, 2008), which, although it uses the R language, is also relevant for statistics in Python. There are also free online resources about statistics, for example, OpenIntro Statistics, available from `www.openintro.org/stat/textbook.php`.